Ravenscar-Java: A High Integrity Profile for Real-Time Java

Jagun Kwon, Andy Wellings, and Steve King

Department of Computer Science

University of York, UK

{jagun, andy, king}@cs.york.ac.uk

York Technical Report YCS 342

May 2002

ABSTRACT

For many, Java is the antithesis of a high integrity programming language. Its combination of object-oriented programming features, its automatic garbage collection, and its poor support for real-time multi-threading are all seen as particular impediments. The Real-Time Specification for Java has introduced many new features that help in the real-time domain. However, the expressive power of these features means that very complex programming models can be created, necessitating complexity in the supporting real-time virtual machine. Consequently, Java, with the real-time extensions as they stand, seems too complex for confident use in high integrity systems. This paper presents a Java profile for the development of software-intensive high integrity real-time systems. This restricted programming model removes language features with high overheads and complex semantics, on which it is hard to perform timing and functional analyses. The profile fits within the J2ME framework and is consistent with well-known guidelines for high integrity software development, such as those defined by the U.S. Nuclear Regulatory Commission.

1. Introduction

Increasingly computers are being used in high integrity real-time systems; that is, systems where failure can cause loss of life, environmental harm, or significant financial penalties. Examples include space shuttles, nuclear power plants, automatic fund transfers and medical instruments. They typically have high development and maintenance costs due to the customised nature of their components. Within such systems, there has been a growing trend to use software, because it provides [Leveson1986, Leveson1991, Parnas+1990, Bowen+1998]:

- improved functionality
- increased flexibility in design and implementation
- reduced production cost
- enhanced management of complexity in application areas.

Java has proved to be an appropriate vehicle for a diverse range of applications including web-based intranets and embedded systems. Its relatively simple linguistic semantics, the adoption of well-understood approaches to managing software

complexity, and support for concurrency seem to have contributed towards its popularity. Initially designed with embedded systems in mind, Java's main goal was to provide engineers with a reliable and cost-effective platform-independent environment.

However, despite all these valuable features, Java has been criticised for its unpredictable performance as well as some security concerns [Appel1999, Azevedo+1999, Amme+2001]. The automatic garbage collection and dynamic class loading mechanisms are often considered problematic, especially under time or performance-critical situations. Moreover, a number of security bugs in the Java virtual machine have been discovered since its first appearance, especially in the bytecode verifiers and Just-in-Time (JIT) compilers [Gong1999, Appel1999]. These fears make Java and its associated technology simply unsuitable for high integrity systems [Kwon+2002].

In recent years, there have been two main activities, initiated by Sun, to address the limitations of Java for real-time and embedded systems. The first is, *the Real-Time Specification for Java* (RTSJ) [Bollella+2000a, Bollella+2000b] which attempts to minimise any modification to the original Java semantics and yet to define many additional classes that must be implemented in a supporting virtual machine. The goal is to provide a predictable and expressive real-time environment. This, however, ironically leads to a language and run-time system that are complex to implement and have high overheads at run-time. Software produced in this framework is also difficult to analyse with all the luxurious features, such as the asynchronous transfer of control (ATC) and dynamic class loading.

The second relevant activitiy is the *Java 2 Platform Micro Edition* (J2ME) [Sun2000]. This essentially defines a three layer architecture:

- a virtual machine layer (usually implemented on top of a host operating system)
- a configuration layer which defines the set of Java language features, a minimum set of virtual machine features and the available class libraries that can be supported by a particular implementation platform (for example, a mobile phone)
- a profile layer which defines a minimum set of Application Programmers Interfaces (APIs) targeted at a particular application domain.

The same configuration layer may support more than one profile, and different configuration layers may support the same profile. A configuration layer, called Connected, Limited Device Configuration (CLDC) has been defined for small, resource-constrained mobile devices (mobile phones, pagers, personal organizers etc.) typically with a memory capacity of up to 512 KB. The K (kilo bytes) virtual machine (KVM) is a virtual machine specifically designed to support the CLDC. The restrictions imposed on the Java language and the virtual machine include [Sun2000]: no support for floating point operations, no native interfaces, no user-defined class loaders, no thread groups or daemon threads, no object finalization, etc. The main motivation for these restrictions is to reduce the size of memory required to implement the virtual machine.

It is clear that the overheads of implementing the RTSJ makes it unsuitable for the CLDC configuration and consequently RTSJ, as it stands, is probably targeted at Java 2 Standard Edition (J2SE) or ideally another configuration (for example, the CDC – Connected Device Configuration) within the J2ME framework. However, a high intergity subset of the RTSJ model would be appropriate for J2ME and it is possible to imagine a high integrity KVM and CLDC along with one or more profiles.

Unfortunately, many language subsets for high integrity systems discourage the use of concurrent activites. For example, Ada is one of the most important programming languages for the high integrity systems application domain. The SPARK subset of Ada [Barnes1997] (which removes many of the language's complicated or advanced features such as tasking, exceptions, overloading etc.) allows programs to be mathematically proven correct. In recent years, advances in real-time systems research and, in particular, in the area of schedulability analysis, have meant that it is now possible to show mathematically that a concurrent program will meet its deadlines. Of course, constraints must be placed on the particular concurrency mechanisms used to ensure predictability. However, it is no longer axiomatic that concurrency should be forbidden or even discouraged.

To encourage the use of concurrency in high integrity real-time systems, the Ada community has developed a subset of the Ada tasking model (including the Real-Time Annex) called the Ravenscar Profile [Burns+1998]. The main aims of the subset are to support a predictable computational model and to enable a small efficient and predictable run-time support system to be produced. The Ravenscar Profile has attracted support from users and compiler (and run-time) vendors, and has become a de facto standard in the high integrity system domain. It will soon be incorporated into the Ada language standard.

Following the philosophy of the Ravenscar profile, we propose a high integrity profile for real-time Java (called Ravenscar-Java) that offers a more reliable and predictable programming environment. In other words, our profile eliminates features with high overheads and complex semantics, so that programs become more analysable and ultimately, more dependable.

This paper is structured as follows: the next section sets the scope and describes the organisation of the profile. In section 3 we show the computational model, before the actual profile is illustrated in detail. Section 5 briefly looks at implementation issues, followed by an example Ravenscar-Java program in Section 6. Related work is considered in Section 7 and the paper concludes with Section 8. The full description of the rules and guidelines of the profile is provided in Appendix A.

2. Scope and Organisation of the Profile

There are many general and sector-specific standards that assist in the construction of high integrity systems (e.g. U.S. DO178B, U.K. DS 00-55, MISRA guidelines, IEC61508). Of particular interest here is the set of software guidelines produced by the U.S. Nuclear Regulatory Commission (NRC) [NUREG/CR-6463] because it is specific to high integrity systems and because it has set up a systematic framework of guidelines by deriving many important attributes from existing standards. There are four top-level attributes:

- Reliability — defined as the "predictable and consistent performance of the software under conditions specified in its design." A key factor in obtaining reliability is to have predictability of the program's execution; in particular: predictability of control and data flow, predictability of memory utilization and predictability of response times.
- Robustness — defined as "the capability of the safety system software to operate in an acceptable manner under abnormal conditions or events." Often called fault tolerance or survivability, this attribute requires the system to cope with both anticipated and unanticipated faults. Techniques such as using replication, diver-

sity and exception handling are commonly used [Burns+2001].

- Traceability — relates to "the feasibility of reviewing and identifying the source code and library component origin and development processes" thus facilitating verification and validation techniques, which are essential aids to ensuring program correctness.
- Maintainability — relates to "the means by which the source code reduces the likelihood that faults will be introduced during changes made after delivery." All the standard software engineering issues apply here such as good readability, use of appropriate abstraction techniques, strong cohesion and loose coupling of components, and portability of software components between compilers and platforms [Sommerville2000].

The report also provides guidelines based on the framework for nine programming languages (including Ada95 and C/C++). Unfortunately, the guidelines do not consider Java.

The goal of this paper is to apply the NRC's framework to Java augmented by the RTSJ. The paper focuses on the *reliability* attribute as the rest of the attributes are concerned with general design decisions that are covered in the software engineering literature. However, we still give several Java specific guidelines in those areas where they have impacts on the RTSJ.

3. Computational Model

The key aim of the Ravenscar-Java profile is to develop a concurrent Java programming model that supports predictable and reliable execution of application programs, thus benefiting the construction of modern high integrity software. Particularly, we follow the philosophy of the Ravenscar profile [Burns+1998] and emphasise the *reliability* attribute of the NRC guidelines. This means that some language features with high overheads and complex semantics are removed for the sake of reliability, and programs are statically analysable in terms of functionality and timeliness before execution. Similarly, the Java virtual machine is also restricted to ensure predictability and efficiency. For example, a Ravenscar-Java VM (RVM) does not support garbage collection.

As in the RTSJ, the Ravenscar-Java profile allows concurrent execution of schedulable objects (threads and event handlers) based on pre-emptive priority-based scheduling. Schedulable objects have to be either periodic or sporadic with minimum inter-arrival times, and the priority ceiling protocol is required to be implemented in the runtime system. This profile facilitates the use of off-line schedulability analysis, which is associated with fixed priority scheduling (e.g. deadline monotonic or rate monotic analysis [Audsley+1993, Liu+1973]).

We assume two execution phases as suggested in [Puschner+2001], i.e. *initialisation* and *mission* phase, as shown below in Figure 1. In the initialisation phase of an application (i.e. the *main*() method and one *RealtimeThread*), all non-time-critical activities and initialisations that are required before the mission phase are carried out. This includes initialisation of all real-time threads, memory objects, event handlers, events, and scheduling parameters[1]. In the mission phase, the application is executed and multithreading is allowed based on the imposed scheduling policy.

[1] This includes loading all the classes needed in the application. In a JIT (Just-In-Time) compilation environment, all loaded classes will be compiled.

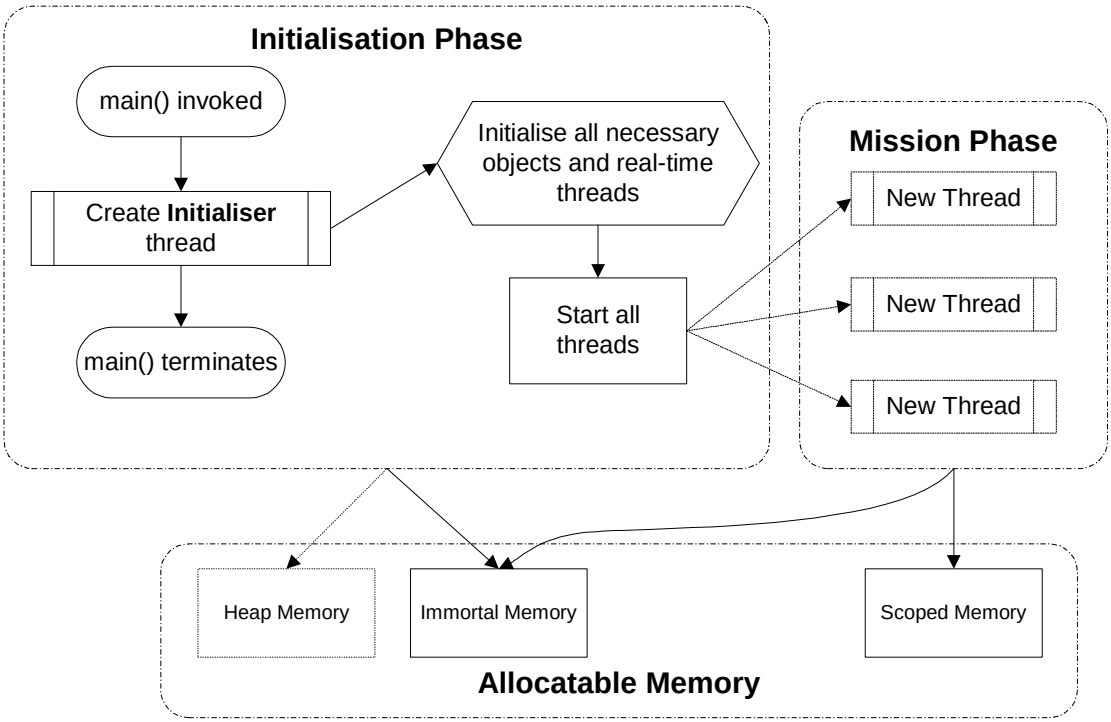

Figure 1. Two execution phases

4. The Profile

The proposed profile can be classified into the following headings, and each of them is expanded below:

- Predictability of memory utilisation
- Predictability of timing
- Predictability of control and data flow.

We separate the rules and guidelines of the profile into the following categories (in Appendix A), and the NRC framework is applied to each of them:

1. Programming in the Large
2. Concurrent Real-Time Programming
3. Programming in the Small.

In the first category, we give guidelines on the use of language features that support high-level decomposition and minimise the complexity of software, which involve object-orientation, and abstract data types. In the second, guidelines on the use of features provided by the RTSJ are presented, whereas in the third we discuss programming issues related to the production of small software components, such as control structures, methods, and expressions. This section summarises the rules and guidelines.

The resulting profile is targeted at a RVM; however, programs written according to the profile are valid RTSJ programs, which will execute without change under a RTSJ virtual machine (although perhaps not within their deadlines).

4.1. Predictability of memory utilisation

This attribute is concerned with ensuring that the software will not access unintended or disallowed memory locations, and ensuring that the use of memory space will be predictable and bounded.

4.1.1. Initialisation and mission phases

When an application is started, its *main*() method will first be invoked by the RVM and the base heap memory area will be used to allocate any objects within the method (as with standard Java).

The main method first creates a new *NoHeapRealtimeThread* with the highest priority in the system. This is required to ensure a well-ordered and controlled execution of the initialisation and mission phases, as illustrated in Figure 2.

```
import javax.realtime.*;
class Main implements Runnable
{
  public static void main(String [] args)
  {
    NoHeapRealtimeThread initializer = new NoHeapRealtimeThread(
          new PriorityParameters(PriorityScheduler.MAX_PRIORITY),
          null, null,
          ImmortalMemory.instance(),
          null,
          new Main());
    initializer.start();
  }

  public void run()
  {
    // initialization phase of the program
  }
}
```

Figure 2. An illustration of the initialisation phase

The new thread must take a reference to the *immortal* memory area, so that all objects and references to other threads and memory objects defined in the *initializer* thread will be safely created and maintained throughout the life of the application. Once all initialisation activities are performed, the thread will allow other threads to execute by invoking the *start*() methods, and terminating itself. To encapsulate this initialisation phase, the Ravenscar-Java profile defines an initializer thread class, shown in Figure 3, which directly extends the **RealtimeThread** class.

```
package ravenscar;
import javax.realtime.*;

public class Initializer extends RealtimeThread
{
  public Initializer()
  {
    super( new PriorityParameters(
      PriorityScheduler.MAX_PRIORITY),
      null, null, ImmortalMemory.instance(),
      null, null);
  }
}
```

Figure 3. **Initializer** class of Ravenscar-Java profile

Now, the application can be created by extending the **Initializer** class in the following way.

```
import ravenscar.*;

public class MyApplication extends Initializer
{
  public void run()
  {
    // logic for initialization
  }

  public static void main (String [] args)
  {
    MyApplication myApp = new MyApplication();
    myApp.start();
  }
}
```

The mission phase begins as soon as the highest priority thread or **Initializer** terminates. From this moment, all application threads will be scheduled and despatched according to the imposed scheduling policy. Threads may only utilise immortal and linear-time scoped memory areas in this phase, unless their logics require access to physical or raw memory areas[2].

4.1.2. Memory Management

To facilitate predictable memory utilisation we define several rules in the three aforementioned areas (see Appendix A for the full list and rationales). The rules place restrictions on, for example, the use of class loaders in the mission phase, on the use of specific memory area objects (and garbage collector), and on recursive method calls. It is also disallowed to create or instantiate any schedulable objects in the mission phase as this will hamper static memory usage analysis.

The heap memory area may or may not exist in a supporting virtual machine. In fact, such memory space can be utilised as part of the whole immortal memory area, since no garbage collection is allowed in the profile.

• Use of immortal memory areas

By definition, objects in an immortal memory area cannot be freed or moved, and all schedulable objects in an application share the same memory area [Bollella+2000a]. Hence, in an attempt to prevent memory exhaustion or corruption, objects (including memory area objects) that are needed for the lifetime of the application should be allocated in the area only in the initialisation phase.

• Use of linear time scoped memory areas

All memory area objects must be created during the initialisation phase (thus, in the immortal memory area), and other objects during the mission phase should make use of **LTMemory** areas. The size of all memory objects must be static and not be

[2] In this paper, we do not attempt to restrict the use of physical or raw memory other than that implied by our restrictions on scoped memory areas. However, a potential implementation of a RVM might apply restrictions for security reasons.

extended in the course of the program. Any other memory area objects defined in the RTSJ are disallowed, and the following simplified classes remain in the profile.

```
package ravenscar;
public abstract class MemoryArea
{
  protected MemoryArea(long sizeInBytes);
  protected MemoryArea(javax.realtime.SizeEstimator size);

  public void enter(java.lang.Runnable logic);
         // throws ScopedCycleException
  public void executeInArea(java.lang.Runnable logic)
         throws InaccessibleAreaException;

  public static MemoryArea getMemoryArea(
                 java.lang.Object object);

  public long memoryConsumed();
  public long memoryRemaining();
  public java.lang.Object newArray(
        java.lang.Class type, int number)
         throws IllegalAccessException, InstantiationException;
        // throws OutOfMemoryError

  public java.lang.Object newInstance(java.lang.Class type)
         throws IllegalAccessException, InstantiationException;
         // throws OutOfMemoryError
  public java.lang.Object newInstance(
               java.lang.reflect.Constructor c,
               java.lang.Object[] args)
         throws IllegalAccessException, InstantiationException;
         // throws OutOfMemoryError;
  public long size();
}

public final class ImmortalMemory extends MemoryArea
{
  public static ImmortalMemory instance();
}

public abstract class ScopedMemory extends MemoryArea
{
  public ScopedMemory(long size);
  public ScopedMemory(SizeEstimator size);

  public void enter();
  public int getReferenceCount();
}

public class LTMemory extends ScopedMemory
{
  public LTMemory(long size);
  public LTMemory(SizeEstimator size);
}
```

Figure 4. Simplified memory area classes

To aid in the production of an efficient RVM and to simplify timing and memory usage analyses, access to **LTMemory** areas must not be nested and **LTMemory** areas must not be shared between **Schedulable** objects.

4.2. Predictability of timing

This attribute focuses on demonstrating that all schedulable objects meet their timing constraints at runtime. The restrictions enforce the computational model given in Section 3 and, thereby, allow schedulability analysis to be performed.

4.2.1. Scheduling and Threading Model

As suggested in the RTSJ, the minimum required scheduling base is by default a fixed-priority pre-emptive scheduler (represented by the **PriorityScheduler** class) that supports at least 28 unique priority levels. The specification also requires that an implementation makes available at least 10 additional native priorities for regular Java threads. However, the profile does not support regular threads by disallowing the use or overriding of the class **java.lang.Thread** to create threads. Therefore, we do not assume any additional native priority levels for regular Java threads. As a result, the supported types of schedulable objects in the profile are

- Periodic threads (see **PeriodicThread** class below), and
- Sporadic event handlers (see **SporadicEventHandler** class below).

The **RealtimeThread** and **AsyncEventHandler** classes are not directly available to the applications programmer, as the former may use the heap memory, whereas the latter hinders accurate timing and memory analyses.

Attributes such as scheduling characteristics and memory areas must be statically allocated to schedulable objects in the initialisation phase, and shall not be changed afterwards, in order to facilitate fixed-priority scheduling algorithms and schedulability analysis. For this purpose, all methods whose names begin with '**set**' (for example, **setReleaseParameters**()) and some with '**get**' are excluded. Thus, the **schedulable** interface is defined as an empty interface, as shown below.

```
package ravenscar;

public interface Schedulable extends java.lang.Runnable
{
}
```

Figure 5. Empty **Schedulable** interface

Only fixed priority-based scheduling is supported by the Ravenscar-Java profile. Furthermore, any subclass of the **Scheduler** including the default **PriorityScheduler** class is not allowed to perform any feasibility checks, leading to the classes in Figure 6. The **PriorityParameters** class also does not contain **setPriority**() method, and the **ImportanceParameters** class is not supported.

```
package ravenscar;

public abstract class Scheduler
{
}

public class PriorityScheduler extends Scheduler
{
  public static final int MAX_PRIORITY;
  public static final int MIN_PRIORITY;
}
```

Figure 6. Simplified **Scheduler** classes

Overall, this approach does not necessitate any dynamic feasibility test and admission control by the RVM at runtime. All schedulability analysis is performed before the initialisation phase of the program.

4.2.2. Use of release parameters

In order to support periodic or sporadic behaviours of real-time threads, the following simplified **ReleaseParameters** class and its subclasses are defined.

```
package ravenscar;

public class ReleaseParameters

{
  protected ReleaseParameters();
}


public class PeriodicParameters extends ReleaseParameters
{
  public PeriodicParameters(AbsoluteTime startTime,
                            RelativeTime period);
  protected AbsoluteTime getStartTime();
  protected RelativeTime getPeriod();
}


public class SporadicParameters extends ReleaseParameters
{
  public SporadicParameters(RelativeTime minInterarrival);
  protected RelativeTime getMinInterarrival();
}
```

Figure 7. **ReleseParameters** and its subclasses

The **AperiodicParameters** class is undefined, as the profile does not support aperiodic activities.

4.2.3. Use of threads

Most of the methods and fields of the original **java.lang.Thread** class are obsolete in the context of the RTSJ and high integrity real-time applications. So, this class is defined as follows[3].

```
package java.lang;
public class Thread implements Runnable
{
  Thread();
  Thread(String name);

  void start();
}
```

Figure 8. Newly defined **java.lang.Thread** class

[3] The profile changes some of the access modifiers of the classes, constructors, and methods in order to ensure they cannot be used directly by the programmer. The changes are always more restrictive and hence programs will always execute on non-Ravenscar implementations.

Along the same lines, the **RealtimeThread** and **NoHeapRealtimeThread** can be defined as:

```
package ravenscar;
public class RealtimeThread extends java.lang.Thread
             implements Schedulable
{
  RealtimeThread(PriorityParameters pp,
         PeriodicParameters p);
  RealtimeThread(PriorityParameters pp,
         PeriodicParameters p, MemoryArea ma);

  public static RealtimeThread currentRealtimeThread();
  public MemoryArea getCurrentMemoryArea();
  void start();
  static boolean waitForNextPeriod();
}

public class NoHeapRealtimeThread extends RealtimeThread
{
   NoHeapRealtimeThread(PriorityParameters pp,
         MemoryArea ma);
   NoHeapRealtimeThread(PriorityParameters pp,
         PeriodicParameters p, MemoryArea ma);

   void start();
}
```

Figure 9. **RealtimeThread** and **NoHeapRealtimeThread** class

• Periodic Threads

Periodic threads transparently invoke the **waitForNextPeriod** method of the **RealtimeThread** class at the end of their main loops to delay until their next periods. Other mechanisms (e.g. **sleep**() method) are prone to have an inaccurate timing model, thus should not be used.

The profile defines an additional class to automate the management of periodic threads, which is shown below.

```
package ravenscar;
public class PeriodicThread extends NoHeapRealtimeThread
{
  public PeriodicThread(PriorityParameters pp,
         PeriodicParameters p, java.lang.Runnable logic);

  public void run();
  public void start();
}
```

Figure 10. **PeriodicThread** class

This class may be utilised as follows. Note that the class assumes the default memory area is the immortal one, and a recovery procedure from a missed deadline can be implemented (if supported by the implementation of the profile).

```
package ravenscar;

public class PeriodicThread extends NoHeapRealtimeThread
{
  public PeriodicThread(PriorityParameters pp, PeriodicParameters p,
                        java.lang.Runnable logic)
```

```
  {
    super(pp, p, ImmortalMemory.instance());
    applicationLogic = logic;
  }

  private java.lang.Runnable applicationLogic;

  public void run()
  {
    boolean noProblems = true;
    while(noProblems) {
      applicationLogic.run();
      noProblems = waitForNextPeriod();
    }
    // A deadline has been missed,
    // If allowed, a recovery routine would be placed here
  }

  public void start()
  {
    super.start();
  }
}
```

Figure 11. An illustration of the **PeriodicThread** class

• Sporadic Activities

Event-triggered activities are supported by means of the **BoundAsyncEventHandler** class. Once an event and its handler are set up, they must remain unchanged permanently. For predictability, it is assumed that each handler is bound to one server thread and each server thread has only one handler bound to it.

Again, we define a new class specifically designed for sporadic activities, as shown below. It is based on the **AsyncEventHandle**r class hierarchy.

```
package ravenscar;
public class AsyncEventHandler implements Schedulable
{
  AsyncEventHandler(PriorityParameters pp,
         ReleaseParameters p, MemoryArea ma);
  AsyncEventHandler(PriorityParameters pp,
         ReleaseParameters p, MemoryArea ma,
         java.lang.Runnable logic);


  public MemoryArea getCurrentMemoryArea();
  protected void handleAsyncEvent();
  public final void run();
}

public class BoundAsyncEventHandler
             extends AsyncEventHandler
{
  BoundAsyncEventHandler(PriorityParameters pp,
         MemoryArea ma, ReleaseParameters p);
  BoundAsyncEventHandler(PriorityParameters pp,
         MemoryArea ma, ReleaseParameters p,
         java.lang.Runnable logic);

  protected void handleAsyncEvent();

}
```

```
public class SporadicEventHandler extends BoundAsyncEventHandler
{
  public SporadicEventHandler(PriorityParameters pri,
                              SporadicParameters spor);
  public SporadicEventHandler(PriorityParameters pri,
                              SporadicParameters spor,
                              java.lang.Runnable);
  public void handleAsyncEvent();
};
```

Figure 12. Event handlers and the **SporadicEventHandler** class

Classes associated with event handlers are shown in Figure 13 below.

```
package ravenscar;
public class AsyncEvent
{
  AsyncEvent();
  void addHandler();
  void fire();
  void bindTo();
}

public class SporadicEvent extends AsyncEvent
{
  public SporadicEvent(SporadicEventHandler handler);
  public void fire();
}

public class SporadicInterrupt extends AsyncEvent
{
  public SporadicInterrupt(SporadicEventHandler handler,
                           java.lang.String happening);
}
```

Figure 13. Associated classes to **SporadicEventHandler** class

Note, that all event handlers are bound to their associated event when the event is created.

• Processing Groups, Overrun and Deadline-miss handlers

Processing groups (i.e. instances of the **ProcessingGroupParameters** class) are not supported in the profile, as they require runtime support for the scheduler to determine the feasibility of the temporal scope of a processing group (which thus hampers static timing analysis). Overrun and deadline-miss handlers are also not required as schedulability analysis has been performed off-line.

4.2.4. Synchronization

The *synchronized* construct in Java provides mutually exclusive access to shared resources or objects, and programmers are always encouraged to use it to avoid data races. However, excessive use of this mechanism may result in poor response time, implying that high priority threads may have to wait until lower ones finish their *synchronized* methods or blocks. Therefore, in order to prevent unbounded priority inversions and possible deadlocks, the priority ceiling protocol (**PriorityCeilingEmulation** class) must be implemented and explicitly used for all objects with synchronized blocks or methods. Furthermore, the profile does not

support *wait*, *notify* and *notifyAll* methods of the object class. All condition synchronization between real-time threads must be via sporadic event handlers as this ensures that the timing properties of the synchronization are properly addressed.

WaitFreeQueues are not required as they are provided in the RTSJ to enable communication between instances of **NoHeapRealtimeThread** and regular Java threads.

4.2.5. Representation of time

Supported representations of time are

- **HighResolutionTime**
- **AbsoluteTime**
- **RelativeTime**
- **RationalTime**.

These classes allow representation of time with up to nanosecond accuracy and precision [Bollella+2000a].

4.2.6. Timer classes

In the presence of the aforementioned classes that offer timely periodic and sporadic behaviours of threads, the **Timer** and its subclasses are not necessary and not available.

4.2.6. Asynchrony

The Asynchronous Transfer of Control (ATC) mechanism is not allowed, as it is one of the most complicated features of the RTSJ and hinders timing and functional analyses [Brosgol+2002].

4.3. Predictability of control and data flow

Predictability of control and data flow is required in order that static analysis techniques can be used to aid programming proof techniques and worst-case execution time analysis. All the rules and guidelines are listed in Appendix A, but noteworthy ones in each of the three areas include

- **Programming in the large**
  - All user-defined classes must include constructors that initialise all internal variables and objects.
  - Dynamic method binding should be minimised. In particular, method overriding and the use of interfaces should be minimised.

- **Concurrent Real-Time Programming**
  - Asynchronous transfer of control (ATC) and any thread aborting mechanisms are disallowed.
  - Use of *wait*, *notify*, and *notifyall* methods is disallowed.

- **Programming in the small**
  - Use of *continue* and *break* statements in loops is disallowed.
  - All constraints, such as one used in a *for* loop, must be static.
  - Compound expressions in parameter passing to methods must be eliminated.

- Expressions whose values are dependent on the order of evaluations should be disallowed.

All the rules and guidelines will facilitate or greatly ease the use of program analysis tools.

5. Implementation Issues

Along the same lines as the profile we present in this paper, it is indispensable to utilise a runtime environment (called RVM in this paper) that has been designed and implemented with highly dependable systems in mind. As mentioned earlier, however, programs based on our profile should be valid RTSJ programs and execute on a standard RTSJ platform with the same functional results.

In addition, tool support is essential to analyse code in terms of functionality and timeliness. A customised tool may be developed that incorporates all the rules and guidelines listed in Appendix A and throughout this paper. Such a tool may also be able to obtain the Worst-Case Execution Time (WCET) and worst-case memory consumption of each thread, thus enabling schedulability analysis [Bernat+2000, Hu+2002]. Standard Java tools or model checkers, such as the ESC/Java [Leino+2000] and Java Pathfinder 2 [Brat+2000, JPF2001] may be used, too.

6. An Example Program

We present a simple and naive traction-control system that senses any difference between the front- and rear-wheel spin speeds, and reduces the engine output if the rear-wheels spin more quickly[4]. There is one periodic thread **SpinMonitor** and one sporadic thread **powerCutHandler**, and as soon as an excessive rear-wheel spin is detected **SpinMonitor** activates **powerCutHandler**. The real application logic is not given as the example is purely intended to illustrate how the profile is used.

```
import ravenscar.*;
import javax.realtime.AbsoluteTime;
import javax.realtime.RelativeTime;
import javax.realtime.ImmortalMemory;

public class TractionController extends Initializer
{
  public void run()                  // Initializer routine
  {
    // powerCutHandler
    SporadicEventHandler powerCutHandler = new SporadicEventHandler (
        new PriorityParameters(15),  // Priority:15
        new SporadicParameters(
          new RelativeTime(333, 0),  // Minimum interarrival time
          5)                         // Buffer size
    )
    {
      public void handleAsyncEvent() // Event handler routine
      {
```

[4] A rear-wheel drive car (e.g. a Formula 1 car) is assumed.

```
        // Logic for handling powerCutEvent event
        // i.e. either cut the engine power or brake appropriate
        // wheels
      }
    };

    final SporadicEvent powerCutEvent=
      new SporadicEvent(powerCutHandler);

    // spinMonitor
    PeriodicThread spinMonitor = new PeriodicThread(
      new PriorityParameters(10),    // Priority:10
      new PeriodicParameters(
        new AbsoluteTime(0, 0),      // Start time
        new RelativeTime(333, 0)     // Period
      ),
      new Runnable() {               // Application logic
        public void run()
        {
          // Logic for checking front and rear wheel spin speeds
          // i.e. obtain sensor readings from front and rear wheels
          // Once any excess of a predefined threshold is detected,
          // fire the following event
            powerCutEvent.fire();
        };
      }
    );

    spinMonitor.start();
  }

  public static void main (String [] args)
  {
    TractionController init = new TractionController();
    init.start();
  }
}
```

7. Related Work

There have been a few subsets or profiles for Java suggested in the literature[5]. None of them, however, are as complete or analytical as the Ravenscar-Java profile described in this paper; they are surveyed below.

7.1. Sequential subset of Java by [Bentley1999]

Bentley [Bentley1999] defines a sequential subset of Java after assessing the language. The subset consists of 21 rules that are effectively derived from [Hutcheon+1992], [MISRA1998] and his assessment. All the rules are categorised into six groups, as shown below with a summary of rules for each group.

• **Rules Concerned With Verification**

[5] In fact, there are subsets of Java defined for other purposes than for use in high integrity systems. For example, in [Drossopoulou+1999] the authors define a series of subsets in order to prove the type soundness of them.

Multithreading is not allowed as it may cause significant difficulties in analysing programs, due mainly to the thread synchronisation mechanisms. In addition, methods and constructors shall not be overloaded.

• **Rules Concerned With Comments**
Comments shall not be nested.

• **Rules Concerned With Predictability**
Variables or objects must be statically initialised (by constructors of appropriate classes), so that no default values are expected. All constraints, such as, those used in *for*-loops, must be static. This will greatly ease various analyses, for example, memory requirement and timing analysis. The *continue* and *break* statements shall not be used, except to terminate the cases of a switch statement, for which a break statement is required for every non-empty case clause. Plus, all switch statements should contain a final default clause. The *return* statement should only appear as the last statement of a method. Further, methods must not have any side effects and not be recursively invoked. The result of a method should never be an unconstrained array type object.

• **Rules Concerned With Constants**
Octal constants (other than zero) shall not be used. Because numbers beginning with zero are treated as octal values in Java, it is easy to make a mistake, e.g. inserting zero before a decimal constant.

• **Rules Concerned With Identifiers**
All identifier names must be unique.

• **Rules Concerned With Operators**
All right-hand operands of the logical operator && and || shall not contain any side effects, since the evaluation and execution of the operands are dependent on the truth-value of the left-hand operand. What is more, assignment operators must not be used in expressions which return Boolean values, for example, in *if ((x=1) != y)*. Bitwise operations, including bitwise shifts, shall not be performed on signed integer types, and the evaluation of integer expressions should not lead to wrap-around.

While this subset will undoubtedly help produce analysable and predictable *sequential* programs, it can be criticised for its restriction on multithreading, one of Java's inherent elements. Without the language-level support for multithreading and all the associated synchronisation mechanisms, Java may not be considered as a great evolution from its predecessors. In addition to this, the subset also fails to address issues on the object-oriented programming model of the language, as well as real-time issues.

7.2. Profile for high integrity Real-Time Java programs [Puschner+2001]

Puschner and Wellings [Puschner+2001] suggest a *Ravenscar*-like profile for the Real-Time Specification for Java [Bollella+2000a], and the following is a brief summary of each of the key areas.

• Threading Model
There are two execution phases, i.e. *initialisation* and *mission* phases. In the initialisation phase, all necessary threads, event handlers, and memory objects are created in a non time-critical manner. No threads will be allowed to start until the top-priority thread with *main()* method finishes its execution. In the mission phase, threads may not change their own or other thread's priority except when forced by the underlying implementation of the priority ceiling protocol. Sporadic or event-triggered activities are implemented as event handlers, and only one handler is allowed per event. All periodic threads must be an instance of *NoHeapRealtimeThread* class and need to invoke *waitForNextPeriod* method to delay execution until the start of their next periods. Asynchronous Transfer of Control (ATC), overrun and deadline-miss handlers, and delay statements are not supported by the profile; nor is dynamic class loading during the mission phase.

• Concurrency
The *synchronized* methods and blocks are the key mechanism for mutual exclusion to shared resources in Java, and the priority ceiling protocol should be implemented in the run-time system in order to avoid deadlocks. For similar reasons, *wait*, *notify*, and *notifyall* are not supported, avoiding any queue management.

• Memory Management and Raw Memory Access
The heap-based garbage collection mechanism of Java is not supported due to its long-debated unpredictability at run-time. Instead, only immortal memory and linear-time scoped memory are supported as defined in the RTSJ. Immortal memory is used by default to create objects during the initialisation phase, but is not allowed for further object creation afterwards. In addition to this, all other memory objects must only be created in the initialisation phase. The RTSJ classes for raw memory access are also supported, so that device drivers, memory-mapped I/O, and other low-level functions can be programmed.

• Time and Clock
All the RTSJ classes for the representation of time and real-time clocks are included while the timer classes are not.

The profile is primarily focused on leaving out complex features of the RTSJ. However, little attention is paid to the Java's sequential language constructs (unlike [Bentley1999]) and object-orientation features that can be problematic in performing various static analyses. Furthermore, the profile is not consistent with the current version of the RTSJ.

7.3. High integrity profile by the J Consortium

A sub-committee has been formed within the Real-Time Java Working Group of the J Consortium to produce a high integrity profile based on the Real-Time Core Extensions [JConsortium2000]. The profile has not been released yet, but according to Dobbing [Dobbing2001] it will resemble the Ravenscar profile for Ada95 [Burns+1998]. It consists of four main themes: partitioning, memory management, concurrency, and error recovery, respectively. Up-coming information will be found at http://www.j-consortium.org/hip/index.shtml.

• Partitioning
The main idea developed from the necessity to isolate critical code and data from non-critical ones by means of firewall, so that less-trusted code will never be able to interfere with high integrity programs. No exchange of objects, as well as dynamic loading across the firewall will be allowed. This idea also extends to the temporal requirements of such software, i.e. temporal firewall, which means deadlines of critical threads must be met.

• Memory Management
The automatic garbage collection is not supported, nor is any memory compaction mechanism. The use of general heap memory is also not allowed. There are three memory allocation strategies, which are

- stack allocation for method local objects that are automatically reclaimed
- fixed size "allocation contexts" for local objects in each thread
- global allocation at initialisation time for immortal objects.

• Concurrency
Three types of priority-based tasks are supported, namely, periodic, sporadic, and interrupt tasks. In addition to these, the profile defines a subclass of the basic *CoreTask* that must explicitly be started by another thread. All threads are created at program start-up, e.g. as part of the initialisation code for classes, and it is not allowed to declare a thread class as an inner class, so that there is no requirement for any implicit *join* interface.

Shared resources and inter-thread synchronisations are managed through *protected objects*, which rely on the underlying implementation of the Priority Ceiling Protocol. However, no mutual exclusion locks or synchronised methods are supported in the profile as they add considerable complexity to program analyses. Further, all the asynchronous thread-to-thread operations, including *stop(), setPriority(), suspend(), resume(),* and event-driven Asynchronous Transfer of Control (ATC) mechanisms, are not permitted, nor are synchronised objects and counting semaphores.

• Error Recovery
The standard exception handling mechanism of Java (i.e. *throw-catch* clause) is maintained. It also supports access to specific physical addresses to allow objects to be mapped, in order to, for example, save program state for fast recovery purposes.

Like the one proposed in [Puschner+2001], this profile is mainly focused on sub-setting the Real-Time Core Extensions [JConsortium2000], but does not address issues on the use of problematic language constructs and object-orientation features of Java.

7.4. Formal subsets by [Drossopoulou+1999]

Drossopoulou *et al.* define three formal subsets of Java, i.e. that of the source language (Java^{s}), high-level representation of bytecode (Java^{b}), and enriched version of Java^{b} (Java^{r}). They present operational semantics, type system, and a proof of type soundness for the subsets.

Java^{s} is a substantial subset of the Java programming language, and it includes some primitive types, interfaces, classes with instance variables and instance methods,

inheritance, hiding of instance variables, overloading and overriding of instance methods, arrays, implicit pointers and the *null* value, object creation, assignment, field and array access, method call and dynamic method binding, exceptions and exception handling [Drossopoulou+1999], as shown below.

Program	::=	*Def**
Def	::=	**class** ClassId ext ClassName **impl** InterfName*
		{*ClassMember**}
	|	**interface** InterfId **ext** InterfName* {*InterfMember**}
ClassMember	::=	*Field* | *Method*
InterfMember	::=	*MethHeader*
Field	::=	*VarType* VarId *;*
Method	::=	*MethHeader MethBody*
MethHeader	::=	(**void** | VarType) MethId ((*VarType ParId*)*) **throws** ClassName*
MethBody	::=	{*Stmts* [**return** *Expr*]] }
Stmts	::=	(*Stmt* ;)*
Stmt	::=	**if** *Expr* **then** *Stmts* **else** *Stmts*
	|	*Var = Expr* | *Expr*.MethName(*Expr**) | **throw** *Expr*
	|	**try** *Stmts* (**catch** *ClassName Id Stmts*)* *finally Stmts*
	|	**try** *Stmts* (**catch** *ClassName Id Stmts*)+
Expr	::=	*Value* | *Var* | *Expr*.MethName(*Expr**)
	|	**new** ClassName() | **new** *SimpleType*([*Expr*])+ ([])* | **this**
Var	::=	Name | *Expr*.VarName | *Expr*[*Expr*]
Value	::=	*PrimValue* | *RefValue*
RefValue	::=	null
PrimValue	::=	*intValue* | *charValue* | *boolValue* | …
VarType	::=	*SimpleType* | *ArrayType*
SimpleType	::=	*PrimType* | ClassName | InterfaceName
ArrayType	::=	*SimpleType*[] | *ArrayType*[]
PrimType	::=	**bool** | **char** | **int** | …

Figure 14. Java^{s} programs [Drossopoulou+1999]

In order to observe run-time behaviours of programs in Java^{s}, they are formally converted into Java^{b} and Java^{r} respectively, which are high-level representations of bytecode with all necessary compile-time type information. Having done this, it is possible to obtain operational semantics of each high-level language construct and prove the soundness of the type system of the source-level subset, Java^{s}.

While these subsets contain many important language constructs of Java that are often omitted in other formal subsets (e.g. exceptions), they still overlook some of Java's inherent features, such as the multithreading and synchronisation models. [Hartel+2001] surveys formal subsets and approaches aimed at improving the safety of Java programs.

8. Conclusions

In this paper, we have presented the *Ravenscar-Java profile*, a high integrity profile for real-time Java. This restricted programming model excludes language features with high overheads and complex semantics, on which it is hard to perform timing and functional analyses. Several classes in the RTSJ are redefined, and a few new classes are added, all resulting in a compact, yet powerful and predictable computational model for the development of software-intensive high integrity real-time systems.

The profile is categorised into three areas, i.e. Programming in the large, Concurrent real-time programming, and Programming in the small. These are then structured based on the guideline framework developed by the U.S. Nuclear Regulatory Commission, which derives many important attributes from existing standards and is specific to high integrity systems. Various rules and guidelines, centred around the *reliability* attribute, are given in each of the three following sub-attributes:

- predictability of memory utilisation,
- predictability of timing, and
- predictability of control and data flow.

A simple example illustrating the use of our profile was also provided in Section 6, before we reviewed four existing subsets of Java or the RTSJ. Most of the subsets, however, overlook some important elements of the language, for example, multi-threading and object-oriented programming model (thus are only concerned with sequential parts), or *vice versa*.

We believe that our profile is expressive enough to accommodate today's demanding requirements for a powerful programming model, yet concise enough to facilitate the implementation of underlying platforms or virtual machines with great ease.

A subset of Java and the RTSJ, along the lines presented in this paper, would be a powerful motivation to develop high integrity systems in Java, rather than in a subset of C, C++ or Ada.

Appendix A. Rules and Guidelines of the High Integrity Profile for Real-Time Java

A.1. Programming in the large

A.1.1. Reliability

• Predictability of memory utilisation

Rule 1. Avoid dynamic class loading in the mission phase
(Also related to rules in Predictability of timing below)
Additional class loading at runtime is seen as overheads to both the virtual machine and the application. Accurate memory and timing analyses are thus impossible and dependent on the location and size of classes, as well as the implemented loading and linking mechanisms.

In order to prevent dynamic class loading, either the virtual machine has to preload all classes that the application utilises, or the application must not be permitted to load any class in the mission phase by restricting the use of the following classes and their subclasses (i.e. user defined class loaders). This can be achieved by employing a simple class hierarchy analyser.

- **java.lang.ClassLoader**
- **java.lang.Class** (*forName*() methods of this class)
- **java.net.URL.ClassLoader**
- **java.security.SecureClassLoader**

• Predictability of control flow

Rule 2. All user-defined classes must include constructors that initialise all internal variables and objects
Java automatically allocates initial values to variables, but programmers must not depend on those as they can be mistakenly used or misinterpreted. Such initial values can also differ from system to system. This rule is equally applied to reference types.

Guideline 1. Use only necessary and analysable classes in the class library for the application domain
To keep the complexity and memory requirement of the application to the minimum, not only should we use absolutely necessary classes in the library, but also the behaviours of such classes must be statically analysable in terms of temporal and functional characteristics.

Guideline 2. Minimise dynamic method binding
(Also related to rules in Predictability of timing below)
Dynamic method binding makes it complex to perform various flow analyses and to obtain the worst-case execution time of a thread. Although there may be only a few choices or branches of methods, the runtime overheads incurred by the virtual machine will be hard to predict and, thus undesirable. Accurate memory requirement analysis can also be difficult when different methods have

different memory utilisation. Therefore, programmers are encouraged not to excessively *override* and *overload* methods with ones that have significantly differing logics and overheads, as this can result in a pessimistic timing analysis. Where the logics are significantly different, the programmer should avoid dynamic dispatching by ensuring that class hierarchies are not passed as parameters to methods.

This guideline equally applies to utilising interface types; the virtual machine needs to resolve a method reference every time it encounters an interface method call at runtime by searching through an interface method table for the method reference since the organisation of the table may vary from class to class that implements the same interface [Venners1999].

Along the same line, *monomorphic* method invocations are greatly recommended wherever possible, in place of *polymorphic* invocations. Code optimisation tools may be used to assist this task.

• **Predictability of timing**

Rule 3. Do not use or override **java.lang.Thread** to create (non real-time) threads

Threads must not be created by instantiating or overriding **java.lang.Thread** class because it provides possibly unsafe asynchronous operations, as well as an inaccurate timing and priority model that are inconsistent with the Real-Time Specification for Java [Bollella+2000a]. It is also impossible to explicitly specify memory requirements for such regular threads. Instead, the **RealtimeThread** class of the specification must be used for all real-time and even non real-time threads (in case of non real-time threads, they must be given a low priority than critical ones and may not invoke **waitForNextPeriod()** method). Ideally, however, applications should make use of the **Initializer**, **PeriodicThread**, and **SporadicEventHandler** classes defined in the profile. Refer to *A.2. Concurrent Real-Time Programming* for a more detailed explanation.

Rule 4. Do not utilise Java classes to schedule threads

(Also related to rules and guidelines in Predictability of memory utilisation and Predictability of control flow above)

Programmers must not make use of the pure Java classes that can be used to schedule threads with an incompatible timing and priority model. Such classes, for example, **java.util.Timer**, **java.util.TimerTask**, and **java.util.Calendar**, ought to be replaced by appropriate counterparts of the Real-Time Specification for Java [Bollella+2000a], which will be discussed in *A.2. Concurrent Real-Time Programming*.

A.1.2. Robustness

• **Controlling use of exception handling**

Guideline 3. Minimise propagation of exceptions

Guideline 4. Localise handling of predefined exceptions

Guideline 5. Handle all user-defined exceptions

Guideline 6. Clearly express and document all user-defined exceptions

(All related to each other)

In Java, when exceptions are not handled locally (i.e. within a *try-catch* block in a method), their enclosing methods will be terminated and returned to the calling method(s). This terminating-and-returning process will continue until an appropriate handler is found. This not only hampers program analysability and adds overheads at runtime, but also could lead to an entire system failure if no proper handler can be located. Hence, every possible effort has to be made to eliminate any uncaught exceptions, i.e. unchecked exceptions and errors.

The *finally* clause may be added to a *try-catch* block, which will always execute before control transfers to a new destination (unless System.exit() method is invoked in the *try* block). It can be used to prevent the propagation of any uncaught exceptions at an outer level of the program or all application threads (i.e. in the initialisation phase) by explicitly transferring control from the finally clause itself to a safe destination, thus abandoning any pending (and possibly disastrous) control transfers that could halt the whole system. A safe destination, which may be part of the initialisation phase, may attempt to restart the application threads that have failed due to an uncaught exception or error, or replace them with threads that have different logics.

• **Checking input and output**

Guideline 7. Methods for input and output should be written defensively

It is a common practice to write a program such that it checks whether or not all input and output values from it are within a legal or specified range. This may prevent some unwanted programming errors. However, this job may be left to program verification tools possibly tailored to a specific application.

A.1.3. Traceability

• **Readability**

See Readability in *A.1.4. Maintainability*.

• **Controlling use of native functions and compiled libraries**

See Guideline 1 in Predictability of control flow above.

A.1.4. Maintainability

• **Readability**

Guideline 8. Comment on the purpose, scope, and date of creation for each object

Guideline 9. Comment on the purpose, and exceptions raising and handling for each method

Guideline 10. Identify dynamic method binding with comments

• **Portability**

Any Java program that supports this profile should be executable on a virtual machine that implements the Real-Time Specification for Java [Bollella+2000a]. This, however, does not imply that such virtual machines will always succeed in providing an accurate, robust and cost-effective runtime base because they may not have been developed with high integrity applications in mind.

A.2. Concurrent Real-Time Programming

A.2.1. Reliability

• Predictability of memory utilisation

Rule 1. Avoid the use of any garbage collection mechanism and heap memory

It has long been argued that the runtime behaviour of the implementation-dependent garbage collector is difficult to predict in terms of its resource (including CPU time) and memory utilisation [Bollella+2000a, Venners1999]. Although there have been some works to improve the situation as in [Kim+1999], it is still challenging to put them into practice. If, however, a predictable garbage collector becomes available, a cautious decision should be made by a reliable organisation after evaluating its usage in high integrity real-time systems.

Without a garbage collector and the use of heap memory area, programmers are able to utilise only *immortal*, *(linear-time) scoped* and *physical* memory areas defined in the Real-Time Specification for Java [Bollella+2000a] to allocate objects. The initialisation phase will use an immortal memory area by default, and object creation in that memory area is allowed only in the initialisation phase (Refer to Rule 2 below).

This rule also renders the use of **java.lang.ref** class obsolete, which allows Java programs to interact with the garbage collector.

Rule 2. Object creation in an *immortal* memory area should be allowed only in the initialisation phase

By definition, objects in an immortal memory area cannot be freed or moved, and all threads in an application share the memory area [Bollella+2000a]. Hence, in an attempt to prevent memory run-out and possible programming errors, this rule is enforced. On the other hand, object creation during the mission phase should make use of **LTMemory** areas.

Rule 3. Do not create or instantiate schedulable objects in the mission phase

Creation or instantiation of schedulable objects, i.e. threads and events, will cause the underlying virtual machine to allocate new memory space and handle a new set of information, which will delay the execution of other threads for an indefinite time. This will hamper low-level memory and timing analyses. Therefore, all schedulable objects must be created in the non-time-critical initialisation phase.

Rule 4. The size of an **LTMemory** area shall not be extended

(Also related to Rule 5 below)

In the RTSJ, the **LTMemory** (or linear time scoped memory) class takes two parameters, one for the initial size, and the other for the maximum size in byte. The two sizes must always be the same in this profile, because any additional memory allocation at runtime may be seen as overheads to the virtual machine, and may not be necessary thanks to static memory analysis.

Rule 5. Do not use nested **LTMemory** areas

The RTSJ allows nested memory scope areas, which can be inefficient and error-prone because the virtual machine needs to check at runtime whether

scopes are properly nested. This runtime check is not desirable, and may have ambiguous time or memory requirements. With this restriction, static analysis of the program can ensure all the assignment rules (for assigning references to objects within different memory areas) are correctly obeyed. That is, an object in the immortal memory/heap must not be able to obtain a reference to an object within a scoped memory area to prevent any dangling references.

Rule 6. LTMemory areas must not be shared between **Schedulable** objects
As with the Rule 5 above, it is difficult to cost-effectively validate if a given scoped memory area is exploited correctly when different schedulable objects share it. Ideally, one thread should have only one dedicated **LTMemory** area to it, or should use the immortal memory area. With this rule enforced, additional overheads of dynamic memory access checking are eliminated, and the virtual machine design and implementation can be significantly simplified.

Rule 7. Create all memory area objects during initialisation phase
All memory areas must be created in the initialisation phase, in order to prevent any runtime overheads for allocating a new memory area.

Rule 8. *Finalizers* must not block (e.g. no *sleep()* method invocation in finalizers)
Generally, finalizers of objects are invoked when the virtual machine detects that there is no more reference to the objects. In the context of the scoped memory area, this process should occur when a memory scope is escaped (i.e. the *reference count* becomes zero), and all the finalizers of the objects in the scope should be invoked. Finalizers of objects allocated in the immortal memory area will only be invoked when the whole application terminates, or there is no runnable *non-demon* thread.

The overheads of finalizers must be taken into account when performing schedulability analysis, and the virtual machine can take some time to free up used memory areas. On the whole, finalizers should be as compact as possible and must not block.

• **Predictability of control flow**

Rule 9. Asynchronous transfer of control (ATC) and any thread aborting mechanisms are disallowed
These features result in high runtime overheads, and obscure static timing and flow analyses. All abnormal conditions that may necessitate the use of ATC must be identified at design stages and prevented by means of off-line analysis and design.

Rule 10. Do not use *wait*, *notify*, and *notifyall* methods
This rule eliminates the need for the whole object queue management in the virtual machine, resulting in more efficient and deadlock-free programs.

Rule 11. Use *synchronized* methods or blocks to access shared objects
These original Java constructs provide mutually exclusive access to shared resources or objects, and programmers are always encouraged to use them to avoid data races. However, excessive use of these mechanisms may result in poor response time, implying that high priority threads that become ready to run

may have to wait until lower ones finish their *synchronized* methods or blocks. Therefore, in order to prevent unbounded priority inversions and deadlocks, the priority ceiling protocol must be implemented in the runtime system and explicitly used for all objects with synchronized blocks or methods.

• **Predictability of timing**

Rule 12. Use only **NoHeapRealtimeThread** class to create periodic threads

In the absence of a garbage collector and heap memory area, the **NoHeapRealtimeThread** class has naturally to be a default framework for modelling periodic threads, which may typically utilise a linear-time scoped memory area. Moreover, only the **waitForNextPeriod** method of that class must be used to delay associated threads because other delay statements (e.g. *sleep*) in Java almost certainly cause difficulties in timing and control flow analyses, and are not compatible with the Real-Time Specification for Java. Preferably, programmers should make use of the **PeriodicThread** class defined in Section 4 of this paper, which automates the timely execution of a given **Runnable** logic.

Rule 13. Use only **BoundAsyncEventHandler** class to model sporadic and event-triggered activities

An instance of the **BoundAsyncEventHandler** class is bound to a dedicated thread permanently, and this way of handling sporadic events eases timing analysis. All event handlers must be initialised and set up with one event each before the mission phase. Once this task is complete, the application must not attempt to rebind the handlers with other event(s), since it will make timing analysis simply impossible. Again, the **SporadicEventHandler** class, defined in Section 4, should preferably be used.

Rule 14. Do not use processing groups, overrun and deadline-miss handlers

The RTSJ allows applications to define processing groups, and have overrun and miss handlers associated with real-time threads. Yet, these are likely to be overheads, as they require runtime support for the scheduler to determine the feasibility of the temporal scope of a processing group. Timing analysis must be statically performed before despatching high integrity software, thus making processing groups and the two sorts of handlers unnecessary.

Guideline 1. Concurrent software design should be as simple as possible

There should be no more threads than necessary and no more thread synchronisations than necessary, so that predictable programs will be produced with low performance penalties. Once an application has entered its mission phase no thread may be created and despatched.

A.2.2. Robustness

See Rule 9 in Predictability of control flow above.

A.2.3. Traceability

No specific rules and guidelines.

A.2.4. Maintainability

• **Readability**

Guideline 2. Identify threads with comments
Guideline 3. Identify memory objects with comments

A.3. Programming in the small

A.3.1. Reliability

• **Predictability of memory utilisation**

Rule 1. Avoid method recursion
Recursive method calls (including mutually recursive calls) can dramatically consume available memory space at runtime, and an erroneous termination condition can cause unbounded recursion. However, this rule may be relaxed if the memory consumption for each method and termination conditions can be formally verified.

• **Predictability of control flow**

Rule 2. Do not use *continue* and *break* statements in loops
The *continue* and *break* statements can be used to jump out of a loop in an uncontrolled manner, which makes static analysis difficult to perform.

Rule 3. Use brackets for every branch in *if-else* statements
The *if-else* statements can have a branch that has a single statement, and such branches do not need brackets. But it can be confusing and lead to programming errors.

Rule 4. All constraints, such as one used in a *for* loop, must be static
This facilitates the prediction and analysis of memory and time requirements of loops prior to program execution. If, however, constraints can change during the course of the program, then at least a tight upper bound must be easily deducible.

Rule 5. Variable declarations must include a static initialisation expression
Java automatically allocates initial values to variables, but programmers must not depend on those as they can be mistakenly used or misinterpreted. Such initial values can also differ from system to system. This rule is equally applied to reference types.

Rule 6. Eliminate compound expressions in parameter passing to methods
Expressions that are used as part of parameters for method calls can easily cause side effects and misinterpretation, leading to unintended behaviours of the program. These particularly include ones with the increment and decrement operators (i.e. ++ and --), which depending on the syntactic position can produce different results. Relational expressions should not appear.

Rule 7. Avoid expressions whose values are dependent on the order of evaluations

In relational operations, the evaluation of the right-hand expression of a logical operator (such as the logical *AND* (&&)) is decided by the truth-value of the left-hand operand. In other words, only if the left-hand expression is considered to be true, will the right-hand one be evaluated. Consequently, it is not recommended for a right-hand expression to contain any operators that can have an influence on the intermediate result of any object or variable, like the assignment operator.

Guideline 1. Use parentheses rather than rely on the default order of precedence
Guideline 2. Use parentheses in bitwise operators
Guideline 3. Use parentheses in comparisons and conditions
Parentheses should be used wherever the meaning of an expression can be vague and needs to be clarified. This will prevent any misinterpretation by programmers.

Guideline 4. Use only one *return* statement per method, preferably at the end of each method
Multiple return statements can make flow and timing analyses difficult or pessimistic to perform.

Guideline 5. Define *defaults* in *switch-case* statements
It is a good programming practice to explicitly state that a *switch-case* statement performs either some given operations or default operations in case there is no condition satisfied.

• **Predictability of timing**

Rule 4 and Guideline 4 in Predictability of control flow remain relevant here.

• **Predictability of mathematical or logical result**

Rule 8. Use the strict floating-point mode (FP-strict) instead of the FP-default mode
The FP-default, introduced in Java 1.2, allows a virtual machine to utilise supported floating-point hardware to speed up its execution, and may store intermediate data in the hardware specific format. However, this way of representing intermediate data is dependent on underlying hardware, and thus hinders portability and even accuracy. Therefore, the original IEEE 754 formats of Java should be used. Nevertheless, this rule may be relaxed if the required precision for a particular application is not important.

Rule 9. Statements that access shared resources or objects must be guarded by a synchronized block or method
Data races can occur if a shared object or variable is accessed by more than one thread, and at least one of them updates the object. The original *synchronized* statement of Java should be used to avoid race conditions, and it is generally the job of the programmer (or a tool) to ensure such statements are safely used and compact enough not to seriously affect the response time of other threads.

Rule 10. Do not use octal constants

Octal constants can be confused with other (decimal) numbers, since any number beginning with a zero will be interpreted as an octal constant by the compiler.

Guideline 6. Remember that integers are truncated when divided
The results of integer divisions are always truncated in Java without any warning such that the precision of the values will be reduced. Floating-point types should be used to prevent any integer truncation.

Guideline 7. Ensure that arithmetic operations produce a result that can be correctly represented
The ranges of values for each type must be considered, and only appropriate values and variables of correct types should be used. Overflow and underflow will never be caught or warned by the compiler, and values may be widened if different types of values and variables are used in expressions.

Guideline 8. Shift operators must be used with caution
The unsigned right shift operator, i.e. >>>, can result in an unexpected value when applied to integer types. That is, Java integer types are all signed and this operator will fill the high-order bits of an integer with zeros, thus possibly changing the sign of that integer value to positive. The left shift operator can also alter the sign of a value.

A.3.2. Robustness

None.

A.3.3. Traceability

- **Controlling use of built-in functions and compiled libraries**

Guideline 9. Minimise the use of native methods (especially without source code)
The use of native methods will certainly hamper portability, and such methods may not have been constructed in the same manner that most other high integrity software is built. In other words, they may inconsistently handle errors, input and output data, and not follow programming rules developed by a governing body or a profile such as this.

A.3.4. Maintainability

- **Readability**

Guideline 10. Blocks should be bounded with brackets
Guideline 11. Minimise use of literals

Acknowledgements

This work has been funded by the EPSRC under award number GR/M94113. The authors gratefully acknowledge the comments of Greg Bollella on an early draft of this paper.

References

[Amme+2001] W. Amme, N. Dalton, M. Franz, and J. Von Ronne, ***SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form***, Accepted for the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation 2001.

[Appel1999] Andrew W. Appel, ***Protection against untrusted code: The JIT compiler security hole, and what you can do about it***, http://www-106.ibm.com/developerworks/library/untrusted-code/, as of January 2001.

[Audsley+1993] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, ***Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling***, Software Engineering Journal, 8(5), 284-92, 1993.

[Azevedo+1999] A. Azevedo, A. Nicolau, and J. Hummel, ***Java Annotation-Aware Just-In-Time (AJIT) Compilation System***, ACM 1999 Java Grande Conference, 1999.

[Barnes1998] J. Barnes, ***High integrity Ada: the SPARK approach***, Addison Wesley, 1997.

[Bentley1999] S. Bentley, ***The Utilisation of the Java Language in Safety Critical System Development***, MSc dissertation, Department of Computer Science, University of York, 1999.

[Bernat+2000] G. Bernat, A. Burns, A. Wellings, ***Portable Worst Case Execution Time Analysis using Java Bytecode***, In Proceedings of the 12th EUROMICRO conference on Real-Time Systems, 2000.

[Bollella+2000a] G. Bollella, et al, ***The Real-Time Specification for Java***, Addison-Wesley, 2000.

[Bollella+2000b] G. Bollella and J. Gosling, ***The Real-Time Specification for Java***, IEEE Computer, Vol. 33, No. 6, June 2000.

[Bowen+1998] J. P. Bowen and M. G. Hinchey, ***High Integrity System Specification and Design***, Springer-Verlag London, 1998.

[Brat+2000] G. Brat, K. Havelund, S. Park, and W. Visser, ***Model Checking Programs***, In IEEE International Conference on Automated Software Engineering (ASE), Sep. 2000.

[Brosgol+2002] B. M. Brosgol, S. Robbins, and R. J. Hassan II, ***Asynchronous Transfer of Control in the Real-Time Specification for Java***, In Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), 2002.

[Burns+1998] A. Burns, B. Dobbing, and G. Romanski, ***The Ravenscar Tasking Profile for High Integrity Real-Time Programs***, In L. Asplund, editor, Proceedings of Ada-Europe 98, LNCS, Vol. 1411, pages 263-275, Berlin Heidelberg, Germany, Springer-Verlag 1998.

[Burns+2001] A. Burns, and A. Wellings, ***Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX***, 3rd Edition, Addison Wesley, 2001.

[Dobbing2001] B. Dobbing, ***The Ravenscar Profile for High Integrity Java Programs?***, ACM Ada Letters, Vol. 21, Issue. 1, March 2001.

[Drossopoulou+1999] S. Drossopoulou and S. Eisenbach, ***Describing the Semantics of Java and Proving Type Soundness***, in LNCS 1523 Formal Syntax and semantics of Java (ed. J. Alves-Foss), Springer-Verlag, Berlin, 1999.

[Gong1999] Li Gong, ***Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation***, Addison-Wesley, 1999.

[Gosling+2000] J. Gosling, B. Joy, G. Steele, and G. Bracha, ***The Java Language Specification***, 2nd Edition, Addison Wesley, 2000.

[Hartel+2001] P. H. Hartel and L. Moreau, ***Formalizing the Safety of Java, the Java Virtual Machine, and Java Card***, ACM Computing Surveys, Vol. 33, No. 4, December 2001.

[Hu+2002] E. Y-S Hu, G. Bernat, and A. Wellings, ***Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems***, In Proceedings of the 5th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), 2002.

[Hutcheon+1992] A. Hutcheon, B. Jepson, D. Jordan, and I. Wand, ***A Study of High Integrity Ada: Language Review***, Technical Report SLS31c/73-1-D, Version 2, York Software Engineering, University of York, July 1992.

[JConsortium2000] J Consortium, ***International J Consortium Specification: Real-Time Core Extensions***, Revision 1.0.14, www.j-consortium.org, September 2000.

[JPF2001] ***Java PathFinder***, http://ase.arc.nasa.gov/visser/jpf/, last accessed in April 2001.

[Kim+1999] T. Kim, N. Chang, N. Kim, H. Shin, ***Scheduling Garbage Collector for Embedded Real-Time Systems***, In Proceedings of the LCTES '99, ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems, 1999.

[Kwon+2002] J. Kwon, A. Wellings, and S. King, ***Assessment of the Java Programming Language for Use in High Integrity Systems***, Technical Report YCS 341, Department of Computer Science, University of York, 2002, available at http://www.cs.york.ac.uk/ftpdir/reports/YCS-2002-341.pdf.

[Leino+2000] K.R.M. Leino, G. Nelson, and J.B. Saxe, ***ESC/Java User's Manual***, SRC Technical Note 2000-002, Compaq Systems Research Center, Palo Alto, CA, 2000. Available at http://www.research.compaq.com/SRC/esc/papers.html, last accessed in July 2001.

[Leveson1986] N. G. Leveson, ***Software Safety: Why, What, and How***, Computing Surveys, Vol. 18, No. 2, ACM, June 1986.

[Leveson1991] N. G. Leveson, ***Software Safety in Embedded Computer Systems***, Communications of the ACM, Vol. 34, No. 2, February 1991.

[Liu+1973] C. Liu and J. Layland, ***Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment***, Journal of ACM, 20(1), 46-61, 1973.

[MISRA1998] The Motor Industry Software Reliability Association, ***Guidelines for the use of the C language in vehicle based software***, The Motor Industry Research Association (MIRA), 1998.

[NUREG/CR-6463] H. Hetcht, M. Hecht, S. Graff, et at, ***Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems***, NUREG/CR-6463, U.S. Nuclear Regulatory Commission, 1997, also available at http://fermi.sohar.com/ J1030/index.htm, last accessed in January 2002.

[Parnas+1990] D. L. Parnas, A. J. van Schouwen, and S. P. Kwan, ***Evaluation of Safety-Critical Software***, Communications of the ACM, Vol. 33, No. 6, June 1990.

[TimeSys2002] TimeSys™, ***Products and Services: Real-Time Java***, available at http://www.timesys.com/rtj/index.html, last accessed in January 2002.

[Puschner+2001] P. Puschner and A. J. Wellings, ***A Profile for High Integrity Real-Time Java Programs***, In Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC), 2001.

[Sommerville2000] I. Sommerville, ***Software Engineering***, 6th Edition, Addison Wesley, 2000.

[Sun2000] Sun Microsystems®, ***Java™ 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices***, White paper, available at http://java.sun.com/j2me/docs/, Sun Microsystems® 2000, last accessed in May 2002.

[Venners1999] B. Venners, ***Inside the Java Virtual Machine***, 2nd Edition, McGraw Hill, 1999.